

A Simple Randomized Algorithm to Compute Harmonic Numbers and Logarithms

Ali Dasdan
 KD Consulting
 Saratoga, CA, USA
 alidasdan@gmail.com

April 24, 2017

Abstract

Given a list of N numbers, the maximum can be computed in N iterations. During these N iterations, the maximum gets updated on average as many times as the N th harmonic number. We first use this fact to approximate the N th harmonic number as a side effect. Further, using the fact the N th harmonic number is equal to the natural logarithm of N plus a constant that goes to zero with N , we approximate the natural logarithm from the harmonic number. To improve accuracy, we repeat the computation over many lists of uniformly generated random numbers. The algorithm is easily extended to approximate logarithms with integer bases or rational arguments.

1 Introduction

We approximately compute the harmonic number and the natural logarithm of an integer as a side effect of computing the maximum of a list of x numbers randomly drawn from a uniform distribution. The key point of this computation is that it basically uses only counting. To improve accuracy, we repeat the computation multiple times and take the average. Using the basic properties of the natural logarithm function, it is simple to extend the algorithm to approximate the logarithms with integer bases or rational arguments. The details follow.

2 Computing the Maximum

```
# compute max(lst)
def compute_max(lst):
    # assume x = length(lst) > 0
    max = -1.0
    for i in lst:      # runs x times
        if i > max:
            max = i    # runs H_x times
    return max
```

Figure 1: A Python function to compute the maximum over a list of x numbers in the interval $[0.0, 1.0)$, where $x > 0$.

Given a list of x numbers in the interval $[0.0, 1.0)$, where $x > 0$, the algorithm (written in the Python programming language) in Figure 1 computes the maximum in x iterations. It is well known that during these iterations, the maximum gets updated H_x times on average, where H_x is the x th harmonic number [1]. The reason for this fact is that in a list of x numbers randomly drawn from a uniform distribution, each number has a probability of $1/x$ of being the maximum.

3 Computing the Harmonic Number and the Natural Logarithm

The x th harmonic number H_x is defined as the series

$$H_x = \sum_{i=1}^x \frac{1}{i} = \ln(x) + \gamma + \epsilon_x \quad (1)$$

where γ is the Euler-Mascheroni constant (roughly equal to 0.57721) and ϵ_x , which is in the interval $(\frac{1}{2(x+1)}, \frac{1}{2x})$, approaches 0 as x goes to infinity [2].

We can rewrite this equation to compute $\ln(x)$ as

$$\ln(x) = H_x - \gamma - \epsilon_x. \quad (2)$$

This means an approximation to H_x can be converted to an approximation to $\ln(x)$.

The algorithm (written in the Python programming language) is given in Figure 2. The inner loop computes the maximum over x uniformly random

```

# compute ln(x) as an average over n iterations
def compute_ln(x, n):
    sum = 0
    sum2 = 0
    random.seed()
    for ni in range(n): # runs n times
        cnt = 0
        max = -1.0 # max of x uniformly random floats in [0.0, 1.0)
        for xi in range(x): # runs x times
            i = random.random()
            if i > max:
                max = i # runs H_x times
                cnt += 1
        sum += cnt # cnt = a new H_x approximation
        sum2 += (cnt * cnt)

    avr = float(sum) / n # average over n H_x's
    sdev = math.sqrt(float(sum2) / n - avr * avr) # standard deviation
    serr = float(sdev) / math.sqrt(n) # standard error of avr
    eps_x = float(1.0) / (2.0 * x) # due to the harmonic series approximation
    gamma = 0.57721 # approximate value of the Euler-Mascheroni constant
    result = avr - gamma - eps_x # ln(x) = H_x - gamma - eps_x
    return result, serr

```

Figure 2: A Python function to approximate the natural logarithm as an average over n iterations of the maximum computation. Each maximum computation goes over x uniformly random numbers and produces a new approximation to the x th harmonic number H_x .

numbers. The outer loop with n iterations is for accuracy improvement; it computes an approximation to H_x every iteration as a side effect of the maximum computation rather than the result of the summation in Equation 1. This H_x computation is an approximation due to two reasons: 1) H_x is never an integer except for $x = 1$ [2], and 2) it is a probabilistic estimate. After these loops exit, the final H_x is set to the average over all these approximations. The natural logarithm is then approximated at the end of this algorithm using Equation 2, where we set ϵ_x to its upper bound of $1/(2x)$.

4 Results

Some results from limited experiments as shown in Figure 3 indicate that the approximation quality is relatively good especially with larger arguments. In this figure, the approximate $\ln(x)$ is the value computed by the algorithm in the previous section and the library $\ln(x)$ is the log function from the Python Math library.

Though we used 1000 repetitions for the results shown, separate limited experiments show that the approximation quality gets better after as low as 10 repetitions.

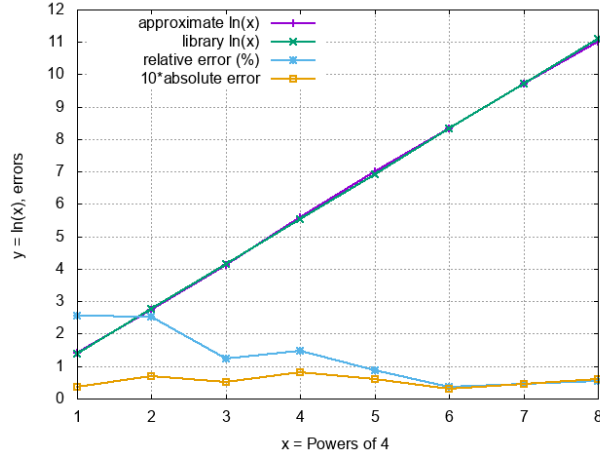


Figure 3: The approximation results on powers of 4 from 1 to 8 with 1000 iterations. The library $\ln(x)$ is `math.log()` from Python’s math library. Both relative and absolute errors are small. Relative error decreases with larger inputs. Note that we multiplied the absolute error by 10 to make it visible in this plot.

5 Pros and Cons

This algorithm to approximate the harmonic number and the natural logarithm takes time proportional to the product of x and n . Even for $n = 1$, the time is linear in x . As such, this is probably not an efficient way of computing the harmonic number or the natural logarithm. What is the use of this algorithm then?

One reason, possibly the main reason, why this algorithm may be interesting is that it approximately computes a function that occurs in its own time complexity analysis. Here the functions are the harmonic number as well as the natural logarithm. Another reason is that this algorithm uses integer arithmetic only except for the final averaging and error computation. Finally, this algorithm is easily parallelizable since the maximum of a list is equal to the maximum over the maximums of parts of the list.

In the technical literature, there are of course many formulas and algorithms for computing both functions [2]. This is expected as the harmonic number and the natural logarithm are so fundamental. This paper is not meant to provide any comparisons with those algorithms or to claim that it is better; it is mainly a fun application on the use of the side effect of a well known and simple algorithm, namely, the maximum computation.

6 Conclusions

We provide a simple algorithm that exploits the time complexity expression of the maximum computation of a list of numbers to approximate the harmonic number and then using it to approximate the natural logarithm. Limited experiments show that the approximations are good with small relative and absolute errors. We hope others may find this algorithm interesting enough to study and potentially improve. At a minimum this paper might hopefully inspire some exercises for students of a basic algorithms book like [1].

References

- [1] T. Cormen, C. S. R. Rivest, and C. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.
- [2] J. Sondow and E. Weisstein. Harmonic number. In *From MathWorld—A Wolfram Web Resource*. Wolfram, Apr 2007. <http://mathworld.wolfram.com/HarmonicNumber.html>.